AD-A220 711

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: NW-LIS-89-10-07	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Fully Testable CMOS Asynchronous Counter		5. TYPE OF REPORT & PERIOD COVERED: Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Gerald R. Carson		8. CONTRACT OR GRANT NUMBER(s): N00014-88-K-0453
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Northwest Laboratory for Integrated Systems, University of Washington, Dept. of Comp. Science, FR-35 Seattle, WA 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA-ISTO, 1400 Wilson Boulevard, Arlington, VA 22209		12. REPORT DATE: October, 1989
		13. NUMBER OF PAGES: 34
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research - ONR, Information Systems Program - Code 1513: CAF, 800 North Quincy Street, Arlington, VA 22217		15. SECURITY CLASS. (of this report): UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Counter, asynchronous, VLSI, testability, circuit design, CMOS

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A testable design for an asynchronous n-bit CMOS counter is presented, with test inputs that provide full coverage for a stuck-at and stuck-open faults. Test time is shown to be O(n), where the counter outputs are not observable, compared to O(n^2) time for a synchronous counter. There are three control signals required for the testable counter as opposed to one reset signal for the base counter. The testable counter incorporates a scan path, utilizing the state storage in the counter cells, whereby the counter is converted into an n-bit

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract (continued from front page)

"master-slave" asynchronous shift register with the counter's "request" input also being used as the shift register input. The only observable outputs are the "acknowledge" and "carry-out" request signals. The counter utilizes two-cycle (transition) signaling and guarantees that new output values are available before "acknowledge" is toggled. Two 16 bit counters, one base design and one scan-based design, are currently in fabrication (2.0 micro n-well CMOS) and will be used to empirically verify the analysis. Splice3 simulations indicate the counter will run at an average speed of approximately 50 MHz. When computed to the base cell the testable design is achieved with a 15% increase in transistor count (from 52 to 60); an increase in chip area of approximately 6%; and a reduction in circuit speed of 7.1% (based on Splice3 simulation).

A Fully Testable CMOS Asynchronous Counter

Gerald R. Carson

Technical Report # 89-10-07

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA
October 1989

DEPARTMENT OF COMPUTER SCIENCE

University of Washington

Seattle 98195

A Fully Testable CMOS Asynchronous Counter

Gerald R. Carson

Technical Report # 89-10-07

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA
October 1989

A report submitted in partial fullfillment
of the requirements for the degree of
Master of Science
University of Washington
1989

A Fully Testable CMOS Asynchronous Counter

by

Gerald R Carson

A report submitted in partial fullfillment
of the requirements for the degree of
Master of Science
University of Washington
1989

ABSTRACT

A testable design for an asynchronous n-bit CMOS counter is presented, with test inputs that provide full coverage for stuck-at and stuck-open faults. Test time is shown to be $O(n)$, where the counter outputs are not observable, compared to $O(n^2)$ time for a synchronous counter. There are three control signals required for the testable counter as opposed to one reset signal for the base counter. The testable counter incorporates a scan path, utilizing the state storage in the counter cells, whereby the counter is converted into an n-bit "master-slave" asynchronous shift register with the counter's **request** input also being used as the shift register input. The only observable outputs are the **acknowledge** and **carry-out** request signals. The counter utilizes two-cycle (transition) signaling and guarantees that new output values are available before **acknowledge** is toggled. Two 16 bit counters, one base design and one scan-based design, are currently in fabrication (2.0 μm n-well CMOS) and will be used to empirically verify the analysis. Splice3 simulations indicate the counter will run at an average speed of approximately 50 MHz. When compared to the base cell, the testable design is achieved with a 15% increase in transistor count (from 52 to 60); an increase in chip area of approximately 6%; and a reduction in circuit speed of 7.1% (based on Splice3 simulation).

Contents

List of Figures

1 Introduction

Asynchronous self-timed systems operate with two control signals: *request* and *acknowledge*. Request signals the system to initiate an action, while acknowledge indicates the action has been completed. This is in contrast to a synchronous system, in which a clock transition initiates an action, which is assumed to be complete some number of clock cycles later. In the asynchronous case, the requestor is in control until the request is generated, at which time control moves to the system receiving the request, and the requestor waits until the corresponding acknowledge signal. Since the requestor is required to wait for the acknowledge, the time taken by the requested system may be arbitrary. It may also vary from request to request, so that performance of an asynchronous system is measured by the average response time, rather than a fixed clock rate.

There are two ways to signal requests and acknowledgments: *two-cycle* or transition signaling and *four-cycle* or level signaling, as discussed in [2]. In four cycle signaling, the quiescent state has request and acknowledge both low. A request event is signaled when the requestor raises the request line. The requested system then performs its action and raises the acknowledge line to signal completion. There then follows a sequence where the requestor lowers the request line to indicate receipt of the acknowledgment, and finally the requested system lowers the acknowledge line to return the interface to the quiescent state. Two-cycle signaling eliminates the time required to return to the low-low state by using the transitions themselves to indicate events. In this mode, the quiescent state is determined when both request and acknowledge are at the same level. A request is signaled by toggling the request line, and later acknowledged by a toggle of the acknowledge line. The state of the system is determined solely by the relative levels of request and acknowledge.

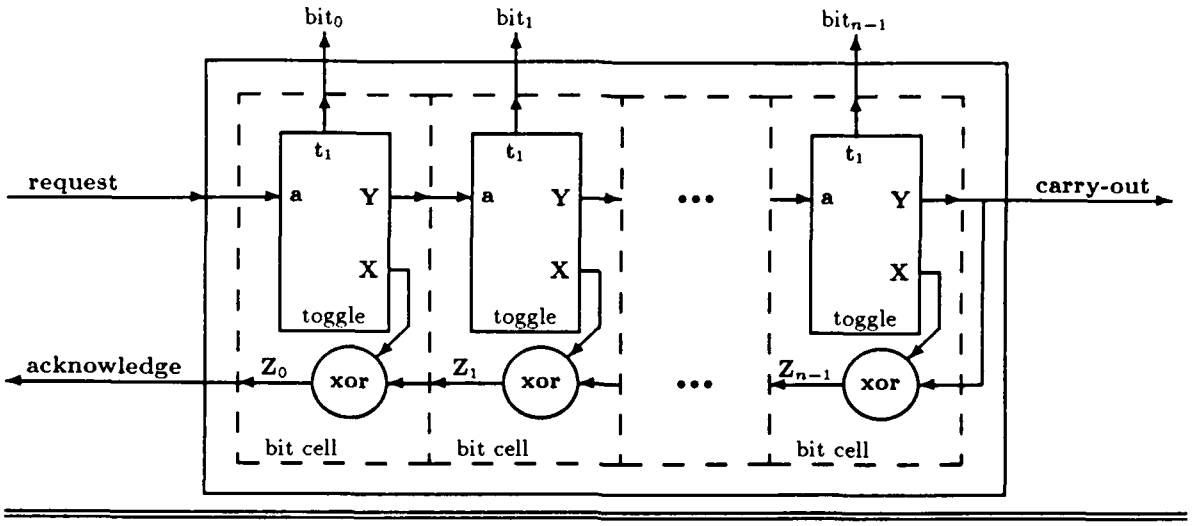

Figure 1: Counter Schematic

The counters discussed in this paper are essentially ripple-carry counters, using two-cycle signaling for control and a *bundled data* protocol, also discussed in [2]. The worst case cycle time for these counters is $n\tau$, (τ is the average delay per cell), while the average cycle time is less than 2τ independent of the number of bits in the counter. The data signals are conveyed by the logic levels on n output lines, while request and acknowledge events are signaled by transitions. The counter guarantees that the data lines are driven to the correct levels before the acknowledge transition. It is left to the user of these signals to assure that delays along the data path are not significantly larger than delays on the acknowledge path. This is the *bundling constraint*. The module assumes the environment is in control whenever request and acknowledge are at the same level. It will then continue to drive out the data lines and the acknowledge line unchanged, until the request line is toggled. At this point, the module has control to update the outputs, and it is assumed that the environment will make no further changes to the request line, and will not sample the counter outputs until the acknowledge line is toggled by the module.

An important property of asynchronous systems is composability, i.e., they may be wired together without regard for timing considerations as long as the same signaling protocols are obeyed. There are many possible types of inter-connection, of which two (*series* and *transition-or*) are used in the asynchronous counter. When two asynchronous modules are wired together in series, the acknowledge signal from the first module becomes a request signal to the second module, and the acknowledge from the second module becomes the acknowledge signal for the combination. The second connection, the *transition-or*, performs the OR function on control signals. It is implemented as an XOR gate. The counter discussed in this paper uses both of these connections. Each bit cell consists of an asynchronous toggle module to generate the correct bit value and carry-out signals, and an XOR gate to combine the acknowledge signal of the toggle with the ackowledge signal from the next cell. These cells are connected in series to form the n-bit counter, as shown in Figure 1.

A question to be considered about asynchronous systems is testability. An n-bit counter is a state machine with 2^n states. Complete functional testing requires exponential time, assuming that the bit values are observable. A recent paper [3] discusses adding a scan path to a simple synchronous counter. with the resulting circuit being fully testable for stuck-at and stuck-open faults in $O(n^2)$ time. We show that similar modifications can be made to the asynchronous counter, and the resulting circuit is fully testable in $O(n)$ time.

2 Fully Testable Asynchronous Counter

Figure 1 shows a block diagram of the counter, and the composition of the individual bit cells. There are n identical cells, each consisting of an asynchronous two-cycle (transition) toggle module and an XOR gate. The counter **request** is wired directly to the input of the low-order toggle cell,

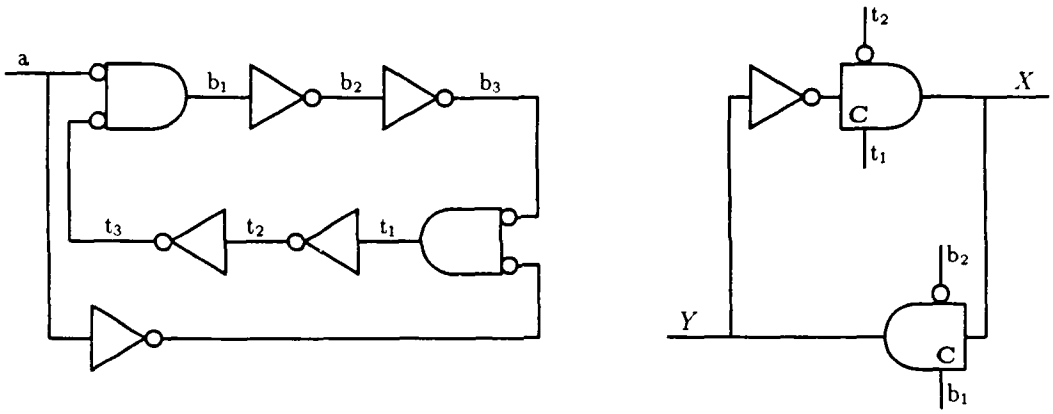

Figure 2: Asynchronous Toggle Schematic

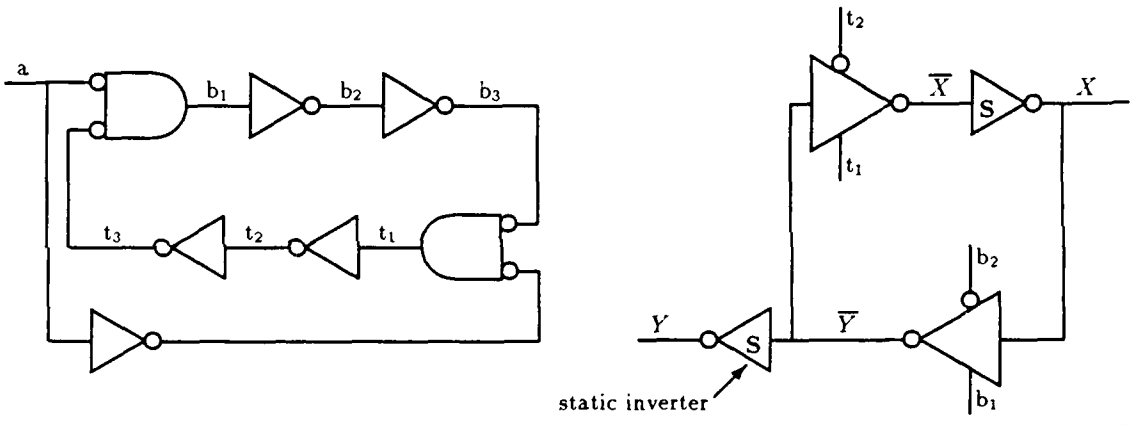

Figure 3: Asynchronous Toggle Schematic – With Clocked Inverters

and the counter **acknowledge** is taken from the XOR gate of the low-order cell. The **carry-out** signal from the high-order bit cell is used as the scan-path output and is routed to an observable pin. It is also connected internally to the last XOR gate, making the counter modulo 2^n.

A transition toggle functions as a divide by two, or alternate output circuit. Figure 2 shows a gate level schematic for a transition toggle There is one input a, and two outputs, X and Y. The cross-coupled NOR loop generates non-overlapping clock signals b_1, b_2, t_1 and t_2 for the oscillator loop containing the X and Y state bits. The oscillator loop consists of two static enabled Muller C-

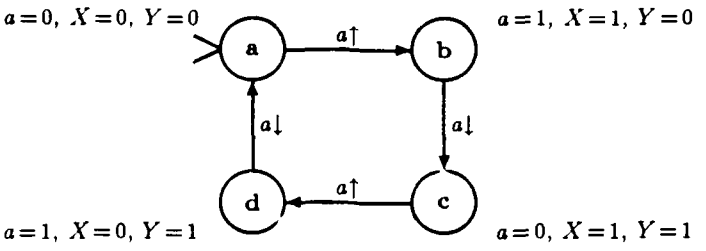

Figure 4: Asyncronous Toggle Transition Diagram

elements and an inverter. The enabled C-element acts as a last-value gate. While the enable signal is high, the output follows the input. When enable goes low, the element remembers the last input value and does not change its output until enable goes high again. C-elements are essentially level-sensitive latches. The control signals for them must be generated by the cross-coupled NOR loop of Figure 2 to ensure that no race conditions exist over the cycle. A static enabled Muller C-element may be built using a clocked inverter followed by a static inverter, leading to the implementation of the transition toggle shown in Figure 3. In operation, the clocked inverters in the C-elements alternately transmit the value of $\overline{Y}$ to X when the input a rises, and the value of X to Y when the input a falls.

The state diagram for a transition toggle is shown in Figure 4. From the reset condition, with $a = X = Y = 0$, the first transition $a\uparrow$ causes a transition $X\uparrow$. The next transition $a\downarrow$ causes a transition $Y\uparrow$, and thereafter, transitions on a cause alternate transitions on X and Y. As used in the counter, the a input is the **request** input to the bit cell; the X output signals **acknowledge** through the XOR gate when the bit value goes from zero to one, ending a carry propagation; and the Y output signals a **carry-out** request to the next cell, when the bit value has changed from one to zero. In this case, the counter acknowledge is generated through the XOR gate from a more

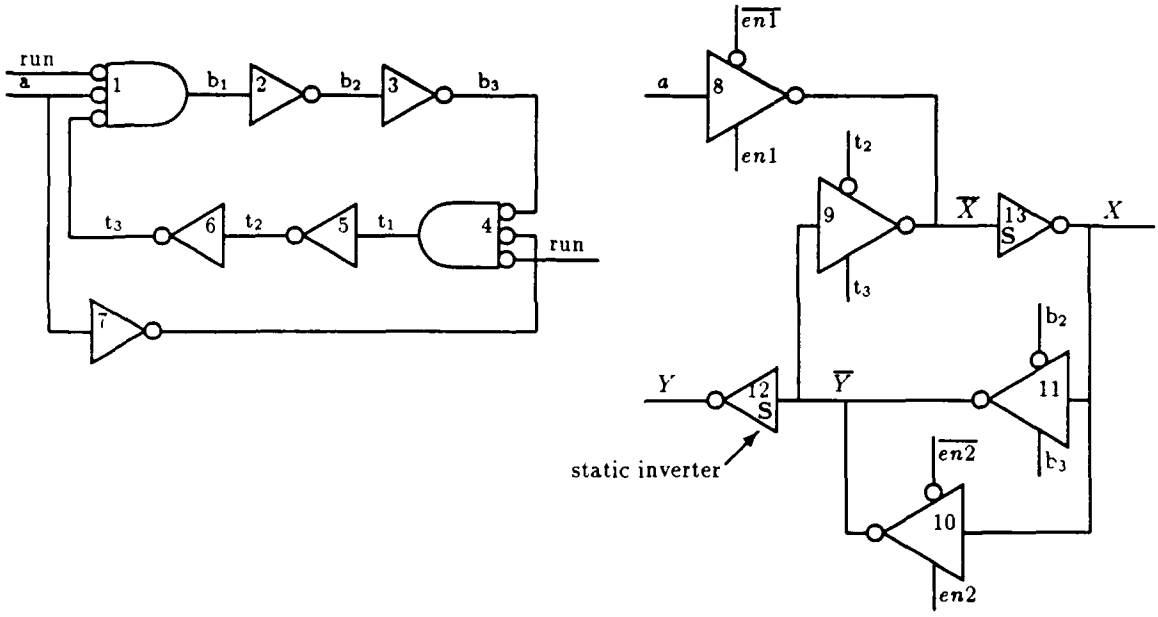

Figure 5: Asynchronous Transition Toggle With Scan Path

significant bit. The internal toggle control signal t_1 always corresponds to the correct bit value and is used as a counter output. The reset signal is not shown. It is active low, and sets the values of X and Y to zero.

All modifications required to add a scan path to the counter occur in the toggle cell, as shown in Figure 5. There are two bits of storage in the toggle cell, the values of X and Y. These values may be isolated by adding a third input, **run**, to the two NOR gates. Asserting **run** turns off both clocked inverters, isolating the X and Y storage. The shift path is then formed with two clocked inverters, added parallel to those in the C-elements. The first, clocked by control signal **en1**, moves the input value a into the X state bit; and the second, clocked by control signal **en2**, moves the current X value into Y. Shifting may now be performed by raising **run** to isolate the cells, and alternately clocking **en1** and **en2** as two non-overlapping signals to move the data. Also, with **run**,

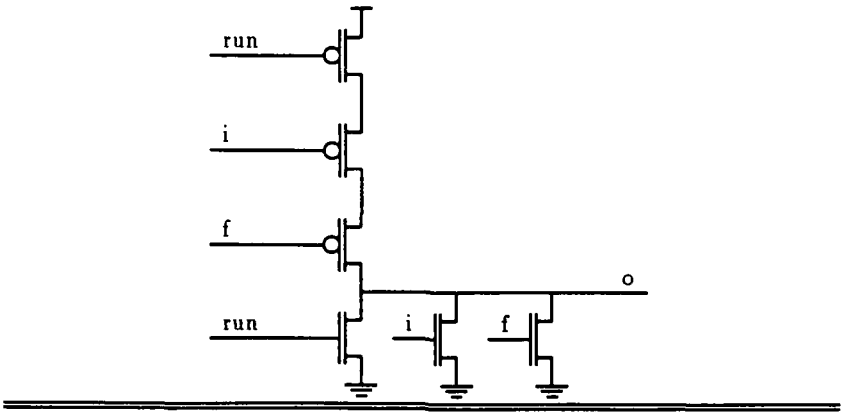

Figure 6: Three-Input CMOS NOR Gate – Transistor Schematic

en1, and **en2** all high, the counter is transparent from **request** through **carry-out**. This allows all X and Y values to be directly set low (or high), eliminating the need for a separate reset signal.

The modifications noted above are sufficient to convert the counter into an n-bit master-slave asnchronous shift register. The shift path has $2n$ bits of storage (2 bits per cell), however only one bit in each cell can be controlled or observed in a shift of n cells. Scan-path operation is discussed in detail in Section 3.2.

Complete fault coverage requires two additional changes in the toggle cell. As discussed in the next section, testing requires charging (discharging) a node, which is then discharged (charged) through the pull-up or pull-down chain to be tested. In the cross-coupled NOR loop, it is not possible to verify the pull-up chains in the inverters generating b_3 and t_3, since they are only used to set up the loop for the next transition. This problem is solved by using the pairs (b_2,b_3) and (t_2,t_3) to enable the clocked inverters, rather than (b_1,b_2) and (t_1,t_2) as in the base toggle design. The cost is one more inverter delay added to the critical path of the toggle cell. The second change is in the three input NOR gates. A transistor level drawing of this NOR gate is shown in Figure 6. In

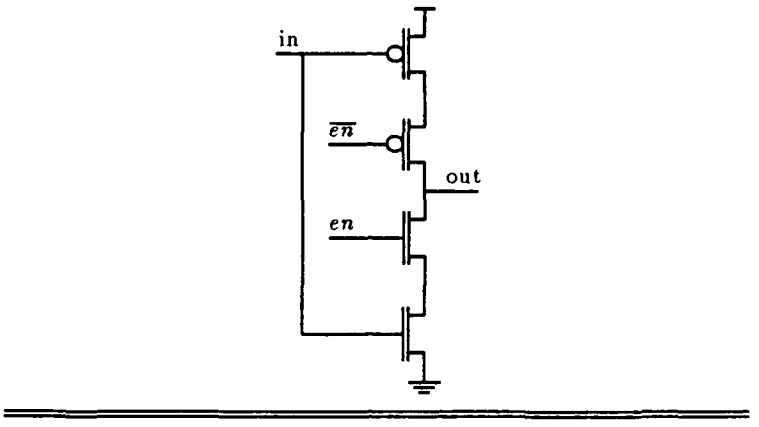

Figure 7: CMOS Clocked Inverter – Transistor Schematic

operation, **run** is always low and the sequence of input transitions is always $i\uparrow$, $o\downarrow$, $f\uparrow$, $i\downarrow$, $f\downarrow$, $o\uparrow$. The output is *always* pulled down through the transistor gated by i, and the only function of the pulldown gated by the feedback term f is to hold the output low for a time equal to the NOR loop delay, which is a few nano-seconds. This function is also performed by charge storage on the node consisting of the NOR gate output and the following inverter input, making the transistor unnecessary. It was eliminated from the testable toggle cell design.

3 Testing The Counter

Circuit testing for stuck-at and stuck-open faults is discussed in [3]. Stuck-at faults result in nodes which are permanently connected to the power supply voltage or ground and never change value. Open faults result in high impedance dynamic states within a pull-up or pull-down path and cause sequential behavior in combinational circuits. Testing for both of these faults involves charging (discharging) each circuit node, followed by discharging (charging) of that same node through the portion of the circuit to be tested. An example is the clocked inverter shown in Figure 7. With **en** high, the inverter is enabled and toggling **in** causes **out** to toggle if the circuit is correct. The

transistors gated by **in** are checked for both open faults and shorts. The transistors gated by **en** are checked only for open faults, since they are always on in the test. Shorts in the transistors gated by **en** are discovered with **en** low and a toggled input. Serial connection of circuit elements simplifies testing, since all elements in the series chain may be tested in parallel [3]. An example is a chain of inverters, where toggling the input must necessarily toggle all internal nodes and the output from the last inverter.

A full test must ultimately verify that every node in the circuit can be pulled high and low. The most straightforward test is a full functional test, which requires that all circuit outputs be observable, and in the case of the counter requires exponential time. An alternative is the addition of a scan path, allowing a known state to be shifted into the counter, and the resulting state after a node is tested to be shifted out. This approach also reduces the number of required observable outputs to the scan path output.

Four test procedures are required to completely test the asynchronous counter with scan path. They are referred to as toggle test, shift test, cycle test, and XOR test; and are discussed in detail below. In these discussions, all references will be to Figure 5, where the gates are numbered from 1–13. When a pull-up or pull-down path is tested, it will be listed in the test description as $\mathbf{u}_i$ or $\mathbf{d}_i$, where i is the gate number in the figure. Each test involves placing the counter in a known state, testing some portion of the circuit, and observing the results either at the **carry-out** pin or the **acknowledge** pin.

3.1 Counter States

The behavior of a cell, bit_i, is totally determined by the state of its internal toggle (see Figure 4), and the value of its input. In the counter, the input to bit_0 is **request**, and the input to bit_i is the value of Y_{i-1}, which is determined by the the state of the toggle in cell bit_{i-1}. The state of the counter may then be completely described by writing down the value of **request** (0 or 1) followed by the state of the toggle internal to each bit cell, from bit_0 through bit_{n-1}. For example, **0aaa** ... is the state immediately after reset.

3.2 Shifting

With signal **run** high, the X and Y values in the toggle cells are isolated, and form a shift chain from **request**, which is the input to bit_0, through **carry-out**, which is the output of the high order cell Y_{n-1}. Three shift operations are possible. Toggling **en1** high and then low is referred to as an **x**-shift. The effect is to copy each cells input into the X state bit, where the input to bit_i is Y_{i-1}, and the input to bit_0 is the **request** signal. This operation destroys the previous contents of X in all cells. The second shifting operation is a **y**-shift, which copies the value of X_i into Y_i for all cells. This is also a destructive operation, in that the former Y values are overwritten. A full shift consists of an **x**-shift followed by a **y**-shift, which treats X and Y as a master/slave pair. The **x**-shiftmoves the new value into X, and the **y**-shift copies this new value into Y. It leaves $X = Y$ in all cells, with X_0 set to **request**, and the old value of Y_{n-2} moved into Y_{n-1} = **carry-out**.

Observing the values of X and Y in each cell requires shifting them to Y_{n-1}, which is the only observable output on the shift path. Y_{n-1} is immmediately observable after a test step. Y_{n-2} is made observable by moving it to Y_{n-1} which requires one full shift (**x**-shift and then **y**-shift),

and in general, *j full shifts* move the contents of Y_{n-j-1} to Y_{n-1}. Note that the very first **x**-shift destroys the contents of all X state bits. Only the Y values are shifted out. To observe the X values resulting from a test, the first operation must be a single **y**-shift, which copies the X values into Y storage, from whence they may be shifted and observed at the **carry-out** pin.

3.3 Toggle Test

With signals **run**, **en1** and **en2** all high, the clocked inverters in the shift path and the static inverters in the cells are all connected in series, and the counter is transparent from **request** through **carry-out**. In this state, toggling the input **request** from low to high to low causes the X and Y values in every cell to be toggled in the same sequence. As in the clocked inverter example, the shift path transistors gated by a and X in gates 8 and 10, and the static inverters in each cell (gates 12 and 13) are completely tested, while the shift path transistors gated by **en1** and **en2** are tested only for open faults. Testing of these transistors for shorts is accomplished in the shift test. The paths tested here include $\mathbf{u}_8$, $\mathbf{u}_{10}$, $\mathbf{d}_8$, $\mathbf{d}_{10}$, $\mathbf{d}_{12}$, $\mathbf{d}_{13}$, $\mathbf{u}_{12}$ and $\mathbf{u}_{13}$.

3.4 Shift Test

The clocked inverters enabled by b_2, b_3, t_2 and t_3 and the correct generation of these signals in the NOR loop are tested by shifting zeros and/or ones through the counter. In general, if the enabling signal is in error due to a problem in the NOR loop, or the gated transistor is shorted, the cell will incorrectly move to another state, which is determined by shifting out the X or Y values. The four possible errors and the corresponding tests are:

1. A cell in state **a**, $X = 0$, $Y = 0$, will move to state **b** if the enabling signal t_3 is high or the pull-down transistor is shorted. This test places all cells in state **a** by performing n full shifts

with **request** low. If this error occurs, some X_i will toggle high. The X values are observed by performing a single **x**-shift to save the X values in the Y storage, and the Y values are shifted out to verify the circuit. The paths tested here include $\mathbf{d}_1$, $\mathbf{u}_2$, $\mathbf{d}_3$, $\mathbf{d}_4$, $\mathbf{u}_5$ and $\mathbf{d}_6$. Also, $\mathbf{d}_9$ is tested for shorted transistors.

2. A cell in state **c**, $X = 1$, $Y = 1$, will move to state **d** if the enabling signal t_2 is low or the pull-up transistor is shorted. This test places all cells in state **c** by performing n full shifts with **request** high. If this error occurs, some X_i will toggle low. The X values are observed by performing a single **x**-shift to save the X values in the Y storage, and the Y values are shifted out to verify the circuit. The paths tested here include $\mathbf{d}_1$, $\mathbf{u}_2$, $\mathbf{d}_3$, $\mathbf{d}_4$, $\mathbf{u}_5$ and $\mathbf{d}_6$. Also, $\mathbf{u}_9$ is tested for shorted transistors.

3. A cell in state **b**, $X = 1$, $Y = 0$, will move to state **c** if the enabling signal b_3 is high or a pull-down transistor ($\mathbf{d}_{10}$ or $\mathbf{d}_{11}$) is shorted. Testing all cells for this error would require placing the counter into state **0bbb**..., which is not possible. This error is tested for in combination with the following.

4. A cell in state **d**, $X = 0$, $Y = 1$, will move to state **a** if the enabling signal b_2 is low or a pull-up transistor ($\mathbf{u}_{10}$ or $\mathbf{u}_{11}$ is shorted. Testing all cells for this error requires placing the counter into state **ddd**..., which is not possible. It is possible however, to place the counter into state **bdbd**... or state **dbdb**..., and check for these last two errors in parallel. First, the counter is placed into the state **0dbdb**... and the odd numbered cells in state **b** are tested for error 3 while the even numbered cells in state **d** are tested for error 4. Then the counter is placed into state **1bdbd**.... and the odd cells are tested for error 4, while the even cells

are tested for error 3.

The counter is placed into state **1bdbd**... by repeating $n/2$ times (One full shift with **request** high followed by one full shift with **request** low), followed by a single **x**-shift with **request** high. To check the results of this test, both the X and Y values must be observed, making it necessary to perform the test twice. The first time, the Y values are shifted out and checked, and the second time, the X values are saved with an **x**-shift, and then shifted out and checked.

The state **1dbdb**... is obtained by repeating $n/2$ times (one full shift with **request** low followed by one full shift with **request** high), followed by a single **x**-shift with **request** low. Again, both the X and Y values must be observed, so the test is performed twice. The first time, the Y values are shifted out and checked, and the second time, the X values are saved with an **x**-shift, and then shifted out and checked. The paths tested here include $\mathbf{d}_1$, $\mathbf{u}_2$, $\mathbf{d}_3$, $\mathbf{d}_4$, $\mathbf{u}_5$, $\mathbf{d}_6$, $\mathbf{d}_9$ and $\mathbf{u}_9$.

3.5 Cycle Test

The remainder of the toggle circuit is verified by simulating normal operation. In normal operation each toggle cycles through states **a**, **b**, **c** and **d**, as shown in Figure 4. The behavior of cell bit_i depends only on its current state and the value of the input, which is the Y value of the preceding cell bit_{i-1}, or the value of **request**. Considering the states of two adjacent cells, a combined state may be written for a pair. There are 16 such pairs, from **aa** through **dd**. Eight of these pairs are stable, in that the output from the left cell in the pair is such that the right cell will remain in its present state. The remaining 8 pairs are unstable, with the input causing the right cell to toggle. Cycle testing of the counter involves shifting in a known sequence of states which includes unstable

sequences, allowing the counter to operate, and then shifting out the resulting values of X and Y. Shifting is performed with signal **run** high, disabling all clocked inverters in the normal operation path. Cycling of the counter cells is then performed by lowering **run** with signals **en1** and **en2** both low. The results are verified by raising **run** and shifting out the X and Y values. Each test sequence must be performed twice: once to shift out the X values; and once to shift out the Y values.

Two state sequences are required to fully test the remainder of the toggle circuit.

- The first sequence is **1ccc**..., which is obtained by setting all X and Y values high. This sequence is unstable, as a cell in state **c** has its Y output high, so that all cells see a high input; while a valid cell in state **c** with high input will cycle to state **d**. Testing then involves setting the counter state, lowering signal **run** to allow the cells to cycle, raising signal **run** to isolate the cells, and finally shifting out the X and Y values. The paths tested are the ones that are active during the transition **c⇒d**, and include $\mathbf{u}_4$, $\mathbf{d}_5$, $\mathbf{u}_6$, $\mathbf{u}_9$ and $\mathbf{d}_{13}$.

- The second state sequence used is **0dbaa dbaa**..., which contains the unstable sequences **0d** and **ad**. In this sequence, the cells in state **d** will cycle to state **a** resulting in a counter state **0abaa abaa**... . The transition involved, **d⇒a** tests paths $\mathbf{u}_7$, $\mathbf{d}_4$, $\mathbf{u}_5$, $\mathbf{d}_6$, $\mathbf{u}_1$, $\mathbf{d}_2$, $\mathbf{u}_{11}$ and $\mathbf{d}_{12}$. The new sequence **0abaa abaa**... contains the unstable sequence **ab**. From this sequence, each cell in state **b** will cycle to state **c**, leading to the counter state **0acaa acaa**... . This transition, **a⇒b** tests paths $\mathbf{d}_1$, $\mathbf{u}_2$, $\mathbf{d}_3$, $\mathbf{d}_7$, $\mathbf{u}_4$, $\mathbf{d}_5$, $\mathbf{u}_6$, $\mathbf{d}_9$ and $\mathbf{u}_{13}$. The state sequence **0acaa acaa**... again contains an unstable sequence **ca**. Here, the cells in state **a** will cycle to state **b**, leading to the final state **0acba acba** which is stable. The third transition involved,

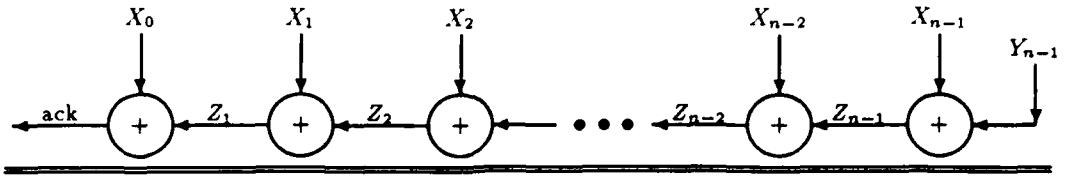

Figure 8: Counter Acknowledge Generation – XOR Chain

b⇒c tests paths $\mathbf{u}_7$, $\mathbf{d}_4$, $\mathbf{u}_5$, $\mathbf{d}_6$, $\mathbf{u}_1$, $\mathbf{d}_2$, $\mathbf{u}_3$, $\mathbf{d}_{11}$ and $\mathbf{u}_{12}$.

This test effectively divides the counter into groups of four cells each, with the cells in state **d** recreating a new carry chain in each group. The test must be repeated four times, using the cyclic permutations of the state sequence **dbaa**..., so that each of the three transitions involved is forced to occur at each bit position in each group, and each individual test must be run twice in order to verify both the X and Y values resulting in each cell.

The basic state pattern **0dbaa dbaa**... is obtained by repeating $n/4$ times (three full shifts with **request** low, followed by one full shift with **request** high), followed by a single **x**-shift with **request** low. The pattern **db** is the key to the $O(n)$ test time of the counter, since this pattern recreats a carry chain, allowing for parallel testing of bit cells. In this sequence, X and Y have different values in each cell, which is made possible by the separate control for **x**-shift and **y**-shift operations.

3.6 XOR Test

The counter acknowledge signal and internal cell acknowledge signals are generated by the XOR gates in the cells. The chain of XOR gates and its inputs is shown in Figure 8. There are two pull-up chains and two pull-down chains in each gate, which must each be tested by setting the gate output to the appropriate value and discharging (charging) the node through the chain to be

tested. This is straightforward in the testable counter since all inputs to the XOR gates are directly settable by the shift chain. There are seven steps involved, all performed with signal **run** high:

1. With **request** low, n full shifts are performed, setting $X = Y = Z = 0$ in all cells.

2. With **request** high, $n-1$ full shifts are performed. After the first full shift X_0 is high, while all other values are unchanged. The low order XOR gate, which generates counter ackowledge, must then toggle its output high for $\overline{X_0}Z_1$. On the second shift, X_1 goes high, and XOR_1 must toggle its output Z_1 high for $\overline{X_1}Z_2$, which then causes XOR_0 to toggle its output low for X_0Z_1. Thereafter, the i^{th} full shift causes all XOR gates from $i-1$ down to 0 to toggle in sequence, always using pullup chain $\overline{X}Z$ and pulldown chain XZ.

3. One **x**-shift is performed to set X_{n-1} high. The high order cell now has X high and Y low, forcing XOR_{n-1} to toggle Z_{n-1} high, and again leading to a toggle sequence down to Z_0.

4. One **y**-shift is performed to set Y_{n-1} high, at which point all values in the cells are high and XOR_{n-1} must toggle Z_{n-1} low, causing one more toggle sequence through the XOR gates. After this step, every pullup chain $\overline{X}Z$, and every pulldown chain XZ has received a toggle test and is verified.

5. At this point, the X and Y values in all cells are high. With **request** low, $n-1$ full shifts are performed. This step is similar to step 2 above, with each shift leading to a toggle sequence through the XOR gates. However, this time the pull-up chain tested is $X\overline{Z}$ and the pull-down chain is $\overline{X}\,\overline{Z}$.

6. One **x**-shift is performed to set X_{n-1} low. This step is similar to step 3 above, leaving Y_{n-1} high, and forcing XOR_{n-1} output Z_{n-1} high, again causing an XOR toggle sequence.

7. One **y**-shift is performed to set Y_{n-1} low. This forces XOR_{n-1} output Z_{n-1} low and leads to the last XOR toggle sequence. After this step, both pull-up chains and both pull-down chains in each cell have been fully tested.

3.7 Test Time

In the case of a synchronous counter with a scan path the test time is $O(n^2)$, since shifting takes $O(n)$ time and the n cells must be tested one at a time. This is necessary due to limitations imposed by the carry chain logic of the counter. There is only one bit of state in each cell of the synchronous counter, and once the carry chain is broken in a cell, it cannot be *restarted* in a higher order cell. In the asynchronous counter there are two bits of state in each cell, both on the scan path, and the added flexibility allows the restarting of the carry chain, thus enabling the parallel testing of cells and an $O(n)$ test time, as follows. The toggle test requires only four steps. All of the remaining tests require shifting values in and/or out of the cells, where shifting takes $O(n)$ time. Since there is a fixed constant number of tests, independent of the size of the counter, the overall test time is $O(n)$.

4 Future Study

An important characteristic of asynchronous systems is composability. The system control path may be formed by the connection of the individual module request/acknowledge signals without regard to timing considerations. It would be desirable to completely fold the scan path into this control path, so that a testable asynchronous system could be composed from testable asynchronous modules without regard to timing considerations or testing considerations. Both would be *automatically* covered by the general design of the control/test interface.

The testable counter, as now designed, has the scan path input merged with the counter request line, but the scan path has two outputs – acknowledge and carry-out. In normal counter operation the control sequence is from request through acknowledge, making it desireable to have carry-out observable at the acknowledge pin when necessary during testing. This implies a multiplexer selecting between carry-out and acknowledge from the low-order cell. The negative aspects of a multiplexer are additional delay on the critical path through the counter in normal operation, and the requirement for a signal to gate the multiplexer.

5 Experimental Results

The 16 bit chip layouts were reduced to 3 bits in order to run Splice3 simulations of both the base design counter, and the counter with scan-path. Three bits provide an excellent estimate of counter performance since each bit cell operates independently, and the performance of the counter is dominated by the operation of the low-order cells. Figure 9 presents the time histories from the simulation for both counters with identical **request** signal inputs. The degradation in performance of the testable design averages 1.2 nano-seconds per bit cell (7.1 %), and is indicated by the later transitions of acknowledge from the scan-path design **ackS** when compared to the base design ackB.

6 Conclusions

Minimal modifications are required to add a scan path to an asynchronous counter, and the resulting circuit is fully testable in linear time. This compares with quadratic time for scan-based synchronous counters with similar observability constraints. Three scan path control signals are required, versus one reset signal for the base design. However, the three signals may be wired together after

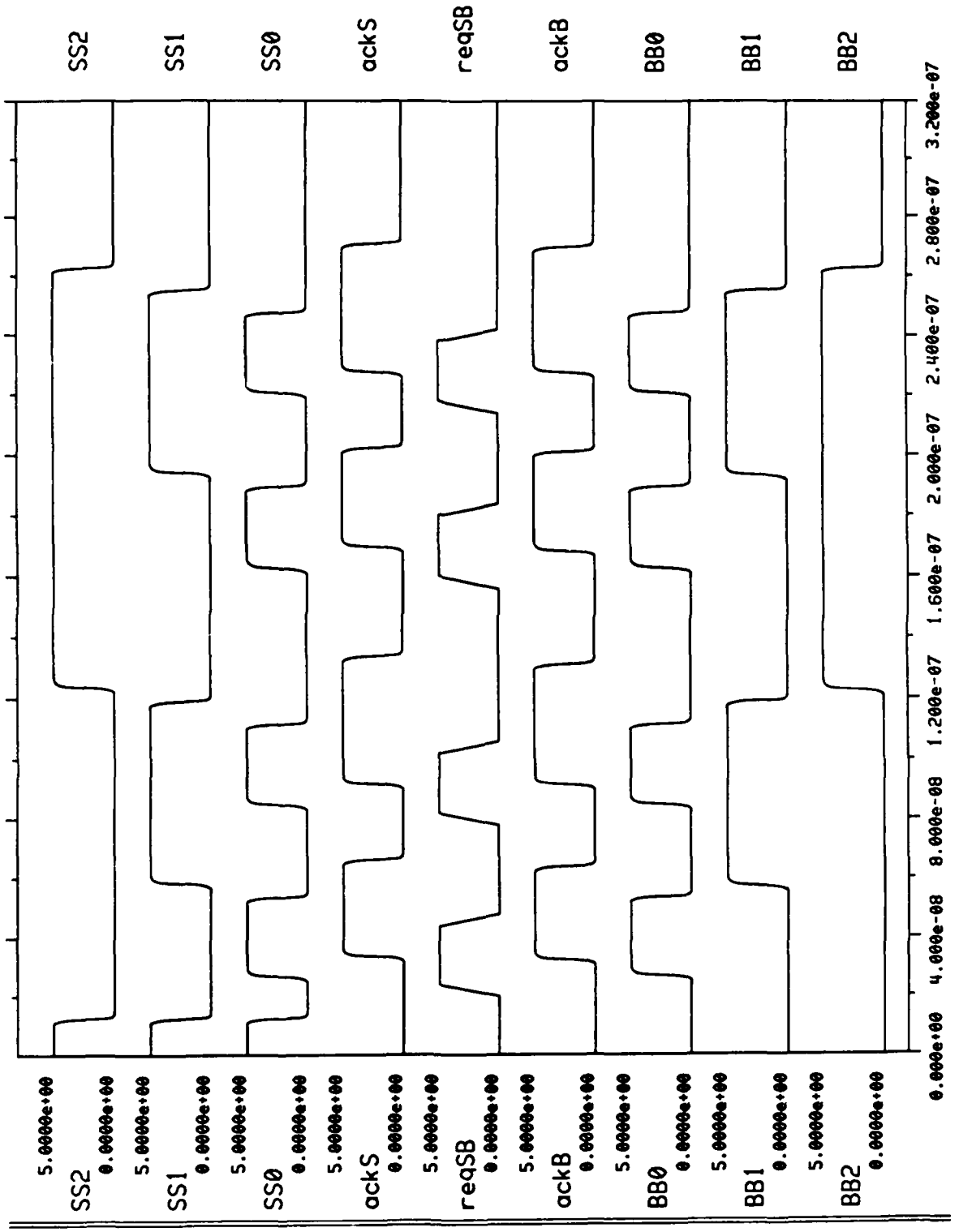

Figure 9: Asynchronous Counters – Splice3 Simulation Results

testing is complete and used as a reset signal for the testable counter. Adding a scan path to the asynchronous counter is actually simplified by the complexity of the base design (compared to a synchronous counter). There are two bits of state per cell in the asynchronous circuit, and the counter may be converted into an n-bit master-slave shift register with the counter's request input being used as the scan-path shift register input. The two state bits in each cell may be set independently, and it is the extra state bit that allows testing in $O(n)$ time since carry propagation chains may be started/restarted anywhere in the counter by appropriately setting the pairs of bits in each cell. The only observable outputs are the acknowledge and carry-out signals. Two 16 bit counters, one base design and one scan based design, have been designed and are currently in fabrication (2.0 n-well CMOS). The base design layout was derived from the scan-design layout in order to provide estimates of area/speed cost of the scan path. The area cost is estimated to be approximately 6%, which compares with an estimate of 15% for the synchronous counter [3]; while the circuit speed of the scan design is reduced by 7.1% (based on Splice3 simulation) from the base circuit.

The counter utilizes two-cycle (transition) signaling and a bundled data protocol. Splice3 simulations indicate the counter will run at an average speed of approximately 50 MHz.

7 Acknowledgements

I would like to thank the LIS and the University of Washington for providing the environment and resources that enabled this project, and to Carl Ebeling for his informative and enlightening seminar on Asynchronous Systems. Also, thanks to Eric Brunvand for providing initial designs and layouts on which the transition toggle was based. Finally, I would like to thank Gaetano Borriello for his ideas, suggestions and support throughout the project.

References

[1] Eric Brunvand, "Parts-R-Us", Technical Report CMU-CS-87-119,
Department of Computer Science, Carnegie Mellon University, 1987.

[2] Robert F. Sproul & Ivan E. Sutherland, "Asynchronous Systems",
Technical Report 4706, 4707, 4708, Sutherland, Sproul and Associates, 1986.

[3] Mehdi Katoozi & Mani Soma, "A Testable CMOS Synchronous Counter",
IEEE Journal of Solid-State Circuits, Vol. 23, No. 5, October 1988.

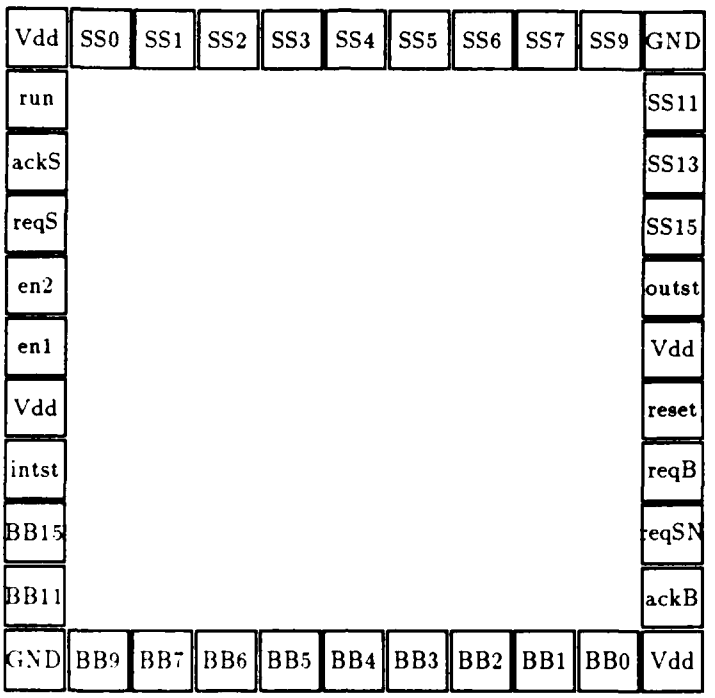

Figure 10: Asyncronous Counter Pin Assignments

A Chip Details

The 40 pin chip (MOSIS tiny-frame) contains two complete 16 bit asynchronous counters, one using the base design and one using the testable scan design. There are insufficient pins on this chip to bring out all bit values in both counters, so only selected high order bits are output. Bit_{15} is output from both counters, and observing this bit will allow speed testing of the counters. The pinouts for the fabricated chip are shown in Figure 10, and the pin functions are listed in Figure 11. Pin names for the base design counter generally contain a **B**, while the testable counter is indicated by S. Two of the pins, **intst** and **outst** are directly connected on the chip to provide an estimate of pad delay. Since the output pad design is inverting, they also provide an inverter for counter testing.

Base Design Counter	
reset	Reset – Active Low
reqB	Request
ackB	Acknowledge
BB0–BB7	Low Order Bits 0-7 – Active High
BB9	Bit Position 9 – Active High
BB11	Bit Position 11 – Active High
BB15	Bit Position 15 – Active High
Testable Counter	
run	Shift Mode – Active High
en1	Enable **x**-shift – Active High
en2	Enable **y**-shift – Active High
reqS	Request
ackS	Acknowledge
reqSN	Carry-Out
SS0–SS7	Low Order Bits 0–7 – Active High
SS9	Bit Position 9 – Active High
SS11	Bit Position 11 – Active High
SS13	Bit Position 13 – Active High
SS15	Bit Position 15 – Active High
Chip Test Pins	
intst	Pad Test Input
outst	Pad Test Output

Figure 11: Asynchronous Counters – Pin Assignments

Figures 12 and 13 present the layouts for the two toggle cells (base design and testable design) using scalable CMOS rules. The layout for the base design was generated by modifying the toggle layout for the testable design, in order to obtain best estimates for area and speed cost of the scan path. The general layout of the 40 pin counter chip is shown in Figure 14.

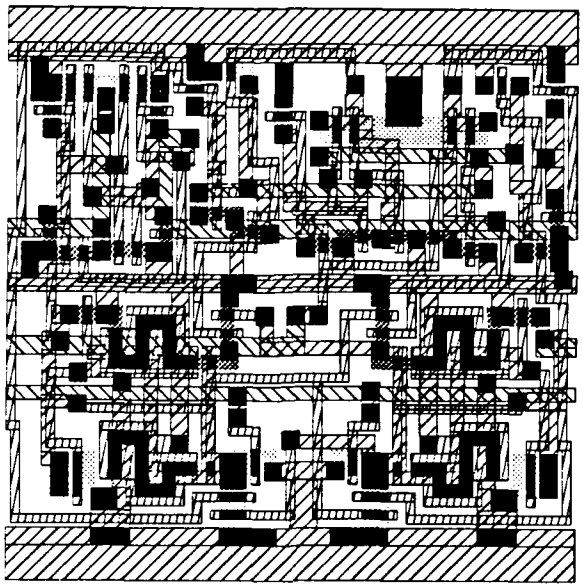

Figure 12: Base Design Asynchronous Toggle - CMOS Layout

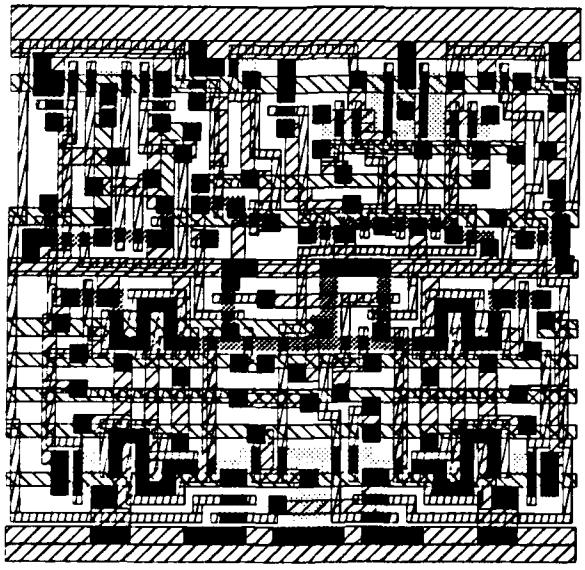

Figure 13: Testable Asynchronous Toggle - CMOS Layout

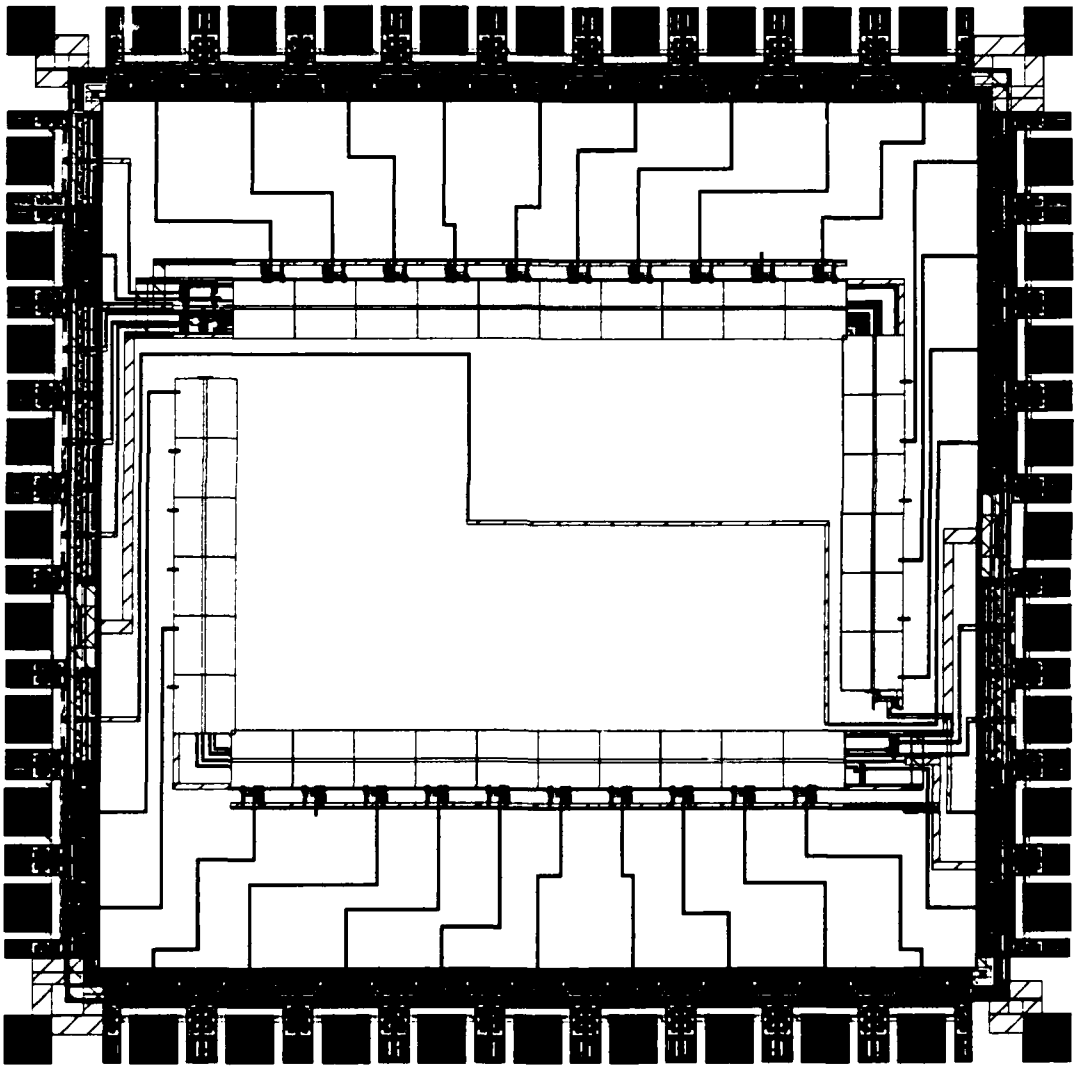

Figure 14: Asynchronous Counter Chip - CMOS Layout

B Counter Test Command File

This appendix contains a listing of the RNL command file which encapsulates the complete test sequence for an N bit testable counter. The test sequence is run with a call to the function runTest.

```
(defun stepp (incr)
  (do ((stop-time (+ incr current-time)) (savex (* current-time 1))
               (n t))
      ((null n))
      (setq n (cond (switch-level (switch-step stop-time))
                    (t (sim-step stop-time))))
      (cond (n (dpy-node-trans n)
               (printf "@ %S\n"
                  (/ (cond (relative-timing (- current-time savex))
                           (t (float current-time)))
                     10)))
            (t nil)))
)

;    Run the counter thru one step
(defun stepc (x)
  (l '(run))
  (stepp incr)
  (h '(run))
  (stepp incr)
)

;    Shift a new value into X
(defun shiftX (x)
  (h '(en1))
  (stepp incr)                          ;Shift into X
  (l '(en1))
  (stepp incr)                          ;New value of X isolated
)

;    Shift a new value into Y
(defun shiftY (y)
  (h '(en2))
  (stepp incr)                          ;Shift into Y
  (l '(en2))
  (stepp incr)                          ;New value of Y isolated
)

;    Full shift step -- X first
(defun shift (x)
  (shiftX nil)
  (shiftY nil)
)

;    Print reqSW and shift next value
(defun showR (a)
  (prReq a)
  (shift nil)
)
```

```
;    Print 4 consecutive values of reqSN
(defun show4R (a b c d)
  (showR d)
  (showR c)
  (showR b)
  (showR a)
)

;    Print 2 consecutive values of reqSN
(defun show2R (a b)
  (shiftV a)
  (shiftV b)
)

;    Shift zero's into all cells
(defun shift0 (N)
  (l '(reqS))
  (repeat i 1 N (shift nil))
)

;    Shift one's into all cells
(defun shift1 (N)
  (h '(reqS))
  (repeat i 1 N (shift nil))
)

;    Shift zero and then one
(defun shift01 (x)
  (l '(reqS))
  (shift nil)
  (h '(reqS))
  (shift nil)
)

;    Shift one and then zero
(defun shift10 (x)
  (h '(reqS))
  (shift nil)
  (l '(reqS))
  (shift nil)
)

;    Print True and Actual value of ackS
(defun prAck (a)
  (printf "$$ ackS=%S - should be %S\n" (node-value 'ackS)  a)
)

;    Print True and Actual value of ackS
(defun prReq (r)
  (printf "$$ reqSN=%S - should be %S\n" (node-value 'reqSN) r)
)

;    Shift Sequence [caaa]
(defun shiftS (x)
  (l '(reqS))
  (shift nil)
  (shift nil)
  (shift nil)
  (h '(reqS))
  (shift nil)
)
```

```
;    Perform Phase 1 of cycleABD Test
(defun cycleABD1 (N mess)
  (unchflag '(reqSN))
  (repeat i 1 (/ N 4) (shiftS nil))
  (l '(reqS))
  (shiftX nil)          ;Sequence [dbaa]4 ==> [acba]4 with input low
  (wr-report)
  (printf "CELLS SHOULD NOW BE [dbaa]4\n")
  (h '(reqS))
  (stepp incr)
  (chflag '(reqSN))
  (l '(run))
  (stepp incr)
  (l '(reqS))
  (stepp incr)
  (h '(run))          ;X values should be [0110]4 Y values should be [0100]4
  (stepp incr)
  (wr-report)
  (printf "CELLS SHOULD NOW BE [acba]4 -- %S\n\n" mess)
)

;    Perform Phase 2 of cycleABD Test
(defun cycleABD2 (N mess)
  (unchflag '(reqSN))
  (shift4S N 1)
  (l '(reqS))
  (shiftX nil)          ;Sequence [adba]4 ==> [aacb]4 with input low
  (wr-report)
  (printf "CELLS SHOULD NOW BE [adba]4\n")
  (chflag '(reqSN))
  (l '(run))
  (stepp incr)
  (h '(run))          ;X values should be [0011]4 Y values should be [0010]4
  (stepp incr)
  (wr-report)
  (printf "CELLS SHOULD NOW BE [aacb]4 -- %S\n\n" mess)
)

;    Perform Phase 3 of cycleABD Test
(defun cycleABD3 (N mess)
  (unchflag '(reqSN))
  (shift4S N 2)
  (shiftX nil)          ;Sequence [aadb]4 ==> [baac]4 with input high
  (wr-report)
  (printf "CELLS SHOULD NOW BE [aadb]4\n")
  (l '(reqS))
  (stepp incr)
  (chflag '(reqSN))
  (l '(run))
  (stepp incr)
  (h '(reqS))
  (stepp incr)
  (h '(run))          ;X values should be [1001]4 Y values should be [0001]4
  (stepp incr)
  (wr-report)
  (printf "CELLS SHOULD NOW BE [baac]4 -- %S\n\n" mess)
)
```

```
;    Perform Phase 4 of cycleABD Test
(defun cycleABD4 (N mess)
  (unchflag '(reqSN))
  (shift4S N 3)
  (h '(reqS))
  (shiftX nil)        ;Sequence [baad]4 ==> [cbaa]4 with input low
  (wr-report)
  (printf "CELLS SHOULD NOW BE [baad]4\n")
  (chflag '(reqSN))
  (l '(run))
  (stepp incr)
  (l '(reqS))
  (stepp incr)
  (h '(run))          ;X values should be [1100]4 Y values should be [1000]4
  (stepp incr)
  (wr-report)
  (printf "CELLS SHOULD NOW BE [cbaa]4 -- %S\n\n" mess)
)

;    Do [caaa] Shift N/4 times
(defun shift4S (N M)
  (repeat i 1 (/ N 4) (shiftS nil))
  (l '(reqS))
  (repeat i 1 M (shift nil))
)

;    Test the xor gates
(defun xor0 (f l)
  (shift nil)
  (prAck f)                          ;ackS should now be f
  (shift nil)
  (prAck l)                          ;ackS should now be l
)

;    Run the Scan Counter for 2 Counts
(defun countS2 (x)
  (h '(reqS))
  (stepp incr)
  (wr-report)
  (l '(reqS))
  (stepp incr)
  (wr-report)
)

;    Run the Base Counter for 2 Counts
(defun countB2 (x)
  (h '(reqB))
  (stepp incr)
  (wr-report)
  (l '(reqB))
  (stepp incr)
  (wr-report)
)
```

```
;     Initialize the Scan Circuit and Test Shift Inverters
(defun initScan (x)
  (setq incr 10000)
  (setq switch-level nil)
  (setq relative-timing t)
  (unchflag '(ackS))
  (chflag '(SS15 SS13 SS11 SS9 SS7 SS6 SS5 SS4 SS3 SS2 SS1 SS0))
  (printf "\nINITIAL TEST\n")
  (wr-report)
  (h '(run en1 en2))
  (l '(reqS))
  (step incr)                          ;X=Y=0 reqSN=0
  (prReq 0)
  (h '(reqS))
  (step incr)                          ;X=Y=1 reqSN=1
  (prReq 1)
  (l '(reqS))
  (step incr)                          ;X=Y=0 reqSN=0
  (prReq 0)
  (l '(en2))
  (stepp incr)                         ;X=Y=0 Isolate X and Y
  (l '(en1))
  (stepp incr)                         ;X=Y=0 Isolate a and X
  (wr-report)
  (unchflag '(SS15 SS13 SS11 SS9 SS7 SS6 SS5 SS4 SS3 SS2 SS1 SS0))
)

;     Initialize the Base Circuit
(defun initBase (x)
  (setq incr 10000)
  (setq switch-level nil)
  (setq relative-timing t)
  (wr-report)
  (printf "\n\nINITIALIZE BASE COUNTER\n")
  (l '(reset reqB))
  (step incr)
  (wr-report)
  (printf "     ackB is %S -- should be 0\n" (node-value 'ackB))
  (h '(reset))
  (step incr)
  (wr-report)
)

;    Test transition from state C to state D
(defun cycleCD (N)
  (printf "\nSTATE TRANSITION TEST\n")
  (shift1 N)
  (stepc nil)
  (wr-report)
  (printf "ALL X VALUES SHOULD BE 0\n")
  (shiftY nil)
  (repeat i 1 N (showR 0))
  (shift1 N)
  (stepc nil)
  (wr-report)
  (printf "ALL Y VALUES SHOULD BE 1\n")
  (repeat i 1 N (showR 1))
)
```

```
;    Perform the shift test
(defun shftScan (N)
  (printf "\nSHIFT TEST -- PART 1\n\n")
  (repeat i 1 N (shift nil))              ;Shift N zero's -- X=0  Y=0
  (repeat i 1 N (showR 0))
  (printf "\nSHIFT TEST -- PART 2\n\n")
  (h '(reqS))
  (stepp incr)
  (repeat i 1 N (shift nil))              ;Shift N one's --- X=1  Y=1
  (repeat i 1 N (showR 1))
  (printf "\nSHIFT TEST -- PART 3\n\n")
  (repeat i 1 (/ N 2) (shift01 nil))      ;X=1  Y=1 in cells 0, 2, 4, etc.
  (l '(reqS))                             ;X=0  Y=0 in cells 1, 3, 5, etc.
  (stepp incr)
  (shiftX nil)                            ;X=0  Y=1 in cells 0, 2, 4, etc.
                                          ;X=1  Y=0 in cells 1, 3, 5, etc.
  (repeat i 1 (/ N 4) (show4R 1 0 1 0))
  (printf "\nSHIFT TEST -- PART 4\n\n")
  (repeat i 1 (/ N 2) (shift10 nil))      ;X=0  Y=0 in cells 0, 2, 4, etc.
  (h '(reqS))                             ;X=1  Y=1 in cells 1, 3, 5, etc.
  (stepp incr)
  (shiftX nil)                            ;X=1  Y=0 in cells 0, 2, 4, etc.
                                          ;X=0  Y=1 in cells 1, 3, 5, etc.
  (repeat i 1 (/ N 4) (show4R 0 1 0 1))
)

;    Test transitions from A-B, B-C, and D-A
(defun cycleABD (N)
  (printf "\nSTATE TRANSITION TEST\n")
  (cycleABD1 N "SHIFT OUT X VALUES")
  (shiftY nil)
  (repeat i 1 (/ N 4) (show4R 0 1 1 0))   ;Print X values
  (cycleABD1 N "SHIFT OUT Y VALUES")
  (repeat i 1 (/ N 4) (show4R 0 1 0 0))   ;Print Y values
  (printf "\nSTATE TRANSITION TEST\n")
  (cycleABD2 N "SHIFT OUT X VALUES")
  (shiftY nil)
  (repeat i 1 (/ N 4) (show4R 0 0 1 1))   ;Print X values
  (cycleABD2 N "SHIFT OUT Y VALUES")
  (repeat i 1 (/ N 4) (show4R 0 0 1 0))   ;Print Y values
  (printf "\nSTATE TRANSITION TEST\n")
  (cycleABD3 N "SHIFT OUT X VALUES")
  (shiftY nil)
  (repeat i 1 (/ N 4) (show4R 1 0 0 1))   ;Print X values
  (cycleABD3 N "SHIFT OUT Y VALUES")
  (repeat i 1 (/ N 4) (show4R 0 0 0 1))   ;Print Y values
  (printf "\nSTATE TRANSITION TEST\n")
  (cycleABD4 N "SHIFT OUT X VALUES")
  (shiftY nil)
  (repeat i 1 (/ N 4) (show4R 1 1 0 0))   ;Print X values
  (cycleABD4 N "SHIFT OUT Y VALUES")
  (repeat i 1 (/ N 4) (show4R 1 0 0 0))   ;Print Y values
)
```

```
(defun xortest (N)
  (printf "\nXOR TEST -- PART 1\n")
  (unchflag '(ackS))
  (wr-report)
  (shift0 N)                        ;X=Y=0 for all cells
  (chflag '(ackS))
  (h '(reqS))
  (stepp incr)                      ;Prepare to shift one's into the counter
  (prAck 0)
  (repeat i 2 (/ N 2) (xor0 1 0));Shift N one's into the counter
  (shift nil)
  (prAck 1)
  (shiftX nil)
  (prAck 0)
  (shiftY nil)
  (prAck 1)
  (wr-report)
  (printf "\nXOR TEST -- PART 2\n\n")
  (unchflag '(ackS))
  (shift1 N)
  (chflag '(ackS))
  (l '(reqS))
  (stepp incr)                      ;Prepare to shift zero's into the counter
  (prAck 1)
  (repeat i 2 (/ N 2) (xor0 0 1));Shift N zero's into the counter
  (shift nil)
  (prAck 0)
  (shiftX nil)
  (prAck 1)
  (shiftY nil)
  (prAck 0)
  (wr-report)
  (unchflag '(ackS))
)

;    Run the Scan Counter for N*2 Counts
(defun countScan (P)
  (l '(reqS))
  (h '(run en1 en2))
  (unchflag '(SS15 SS13 SS11 SS9 SS7 SS6 SS5 SS4 SS3 SS2 SS1 SS0))
  (stepp incr)
  (l '(en1 en2))
  (stepp incr)
  (l '(run))
  (stepp incr)
  (chflag '(ackS SS15 SS13 SS11 SS9 SS7 SS6 SS5 SS4 SS3 SS2 SS1 SS0))
  (wr-report)
  (repeat i 1 P (countS2 nil))
)
```

```
;    Run the Base Counter for N*2 Counts
(defun countBase (P)
  (l '(reqB reset))
  (unchflag '(BB15 BB11 BB9 BB7 BB6 BB5 BB4 BB3 BB2 BB1 BB0))
  (stepp incr)
  (h '(reset))
  (stepp incr)
  (chflag '(ackB BB15 BB11 BB9 BB7 BB6 BB5 BB4 BB3 BB2 BB1 BB0))
  (wr-report)
  (repeat i 1 P (countB2 nil))
)

;    Run the Complete Test Sequence
(defun runTest (N C)
  (def-report '(" " (vec RA) (vec SS) (vec XS) (vec YS) ))
  (printf "\n\n ---------------- BEGIN SCAN TEST\n\n")
  (initScan nil)
  (shftScan N)
  (cycleCD N)
  (cycleABD N)
  (xortest N)
  (printf "\n\n ---------------- END SCAN TEST\n\n")
  (def-report '(" " (vec RA) (vec SS) ))
  (printf     " ---------------- FUNCTION TEST -- COUNT FROM ZERO\n\n")
  (countScan C)
  (printf "\n\n ---------------- END OF PARTIAL FUNCTION TEST\n")
  (chflag '(ackB))
  (def-report '(" " (vec RB) (vec BB) (vec XB) (vec YB) ))
  (printf "\n   ---------------- BEGIN BASE TEST\n\n")
  (chflag '(BB15 BB11 BB9 BB7 BB6 BB5 BB4 BB3 BB2 BB1 BB0))
  (initBase N)
  (printf "\n\n ---------------- FUNCTION TEST BASE -- COUNT FROM ZERO\n\n")
  (def-report '(" " (vec RB) (vec BB) ))
  (countBase C)
  (printf "\n\n ---------------- END FUNCTIONAL TEST - BASE CIRCUIT\n\n")
)
```